UM FRAMEWORK PARA O DESENVOLVIMENTO DE PROGRAMAS DE SIMULAÇÃO DISTRIBUÍDA

Edmilson Marmo Moreira
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-903 – Itajubá-MG
edmarmo@unifei.edu.br

Otávio Augusto Salgado Carpinteiro
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-903 – Itajubá-MG
otavio@unifei.edu.br

Enzo Seraphim
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-903 – Itajubá-MG
seraphim@unifei.edu.br

Thatyana de Faria Piola Seraphim
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-903 – Itajubá-MG
thatyana@unifei.edu.br

Liverson Batista da Cruz
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-903 – Itajubá-MG
Liver10son@yahoo.com.br

RESUMO

Este artigo apresenta um *framework* para facilitar o desenvolvimento de programas de simulação que utilizem uma infra-estrutura distribuída através de uma máquina paralela ou um sistema distribuído. O *framework* foi projetado para trabalhar com protocolos otimistas e conservativos, além da flexibilidade de se utilizar as bibliotecas de troca de mensagens PVM (*Parallel Virtual Machine*) ou MPI (*Message Passing Interface*). Esta flexibilidade permite melhorar o desempenho das aplicações através do desenvolvimento de mecanismos para troca dinâmica entre os protocolos.

PALAVRAS CHAVE. Simulação distribuída. *Framework*. Protocolos de sincronização. Simulação.

ABSTRACT

This paper presents a framework to facilitate the development of simulation programs that use a distributed infrastructure over a parallel machine or a distributed system. The framework is designed to work with conservative and optimistic protocols. In addition, the framework has flexibility to use the passing message libraries PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). This flexibility improves the performance of applications through the development of mechanisms for dynamic exchange of protocols.

KEYWORDS. Distributed Simulation. Framework. Synchronization Protocols. Simulation.

1. Introdução

A simulação é bastante abrangente e tem sido largamente empregada em diversas áreas, destacando-se as de meteorologia, saúde, engenharia, indústria, militar, redes de telecomunicações e de longa distância, entre outras (TAYLOR et al., 2002; JI et al., 2004). A Simulação também tem sido utilizada na avaliação de desempenho de sistemas computacionais, devido à sua flexibilidade e baixo custo (PEGDEN; SHANNON; SADOWSKI, 1995).

Por sua vez, a simulação distribuída (FUJIMOTO, 2000), tem sido utilizada para diminuir o tempo gasto na execução dos programas de simulação. Entretanto, o desenvolvimento de programas paralelos e/ou distribuídos para este tipo de aplicação não é uma tarefa simples. Isto ocorre devido à multiplicidade de recursos envolvidos e a necessidade de sincronizar os processos que compõem a aplicação. Esta tarefa é complexa devido à inexistência de relógios globais de referência e a multiplicidade de recursos de *hardware* e *software*. A sincronização dos processos é necessária para evitar que ocorram erros de causa e efeito, que consistem em inconsistências relacionadas à ordem cronológica de processamento de tarefas.

O problema da sincronização dos processos de um programa de simulação tem sido tratado através do uso de protocolos de sincronização que garantem a consistência dos resultados. Os protocolos são classificados em conservativos e otimistas (FUJIMOTO, 2003), devido à diferença no tratamento dos erros de causa e efeito. Os protocolos conservativos previnem a ocorrência destes erros enquanto os protocolos otimistas utilizam algum mecanismo de correção quando eles ocorrem. Porém, mesmo com o desenvolvimento de tais protocolos, não é uma tarefa fácil aplicá-los em modelos para reduzir o tempo de simulação devido à complexidade encontrada em suas estruturas. Neste contexto, este artigo apresenta um *framework* para o desenvolvimento de aplicações de simulação distribuída, possuindo um conjunto de classes personalizadas e abstratas desenvolvidas para solucionar diversos problemas encontrados em ambientes distribuídos como, por exemplo, a escolha do protocolo e a possibilidade de trocar dinamicamente o protocolo durante a simulação.

Este artigo esta estruturado da seguinte forma: a seção 2 apresenta uma breve revisão da literatura e a seção 3 discute a modelagem do *framework*. A seção 4 apresenta uma discussão sobre “troca dinâmica de protocolos” e, finalmente, a seção 5 conclui o artigo.

2. Trabalhos relacionados

Existem vários protocolos de sincronização para programas de simulação distribuída. De forma geral, os conservativos seguem o paradigma imposto pelo protocolo CMB (*Chandy-Misra-Bryant*) (BRYANT, 1977; CHANDY;MISRA, 1979) e suas variantes (MISRA, 1986; LIU; TROPPER, 1990; CAI; TURNER, 1990; BOUKERCHE; TROPPER, 2001) que consideram o tratamento de *deadlocks*, número de mensagens no sistema e a importância de se determinar um intervalo seguro para o processamento dos eventos (*lookahead*). No entanto, a rigidez na forma de sincronização desses protocolos dificulta à obtenção de bons resultados. Na abordagem otimista, o *Virtual Time* (JEFFERSON, 1985) tem sido a base para o sincronismo, apesar das tentativas de reduzir o custo computacional na execução do procedimento de *rollback*, através da diminuição do otimismo da aplicação (SRINIVASAN; REYNOLDS, 1998; ISKRA; ALBADA; SLOOT, 2003) e melhorias nas técnicas de gerenciamento da memória, uma vez que este tipo de abordagem necessita que os processos armazenem seus estados para viabilizar o procedimento de *rollback* (ZENG; CAI; TURNER, 2004).

Em relação aos protocolos otimistas, existem, ainda, protocolos que procuram garantir o sincronismo dos processos lógicos utilizando abordagens diferentes como, por exemplo, o protocolo *Rollback* Solidário (MOREIRA, 2005). Este protocolo está fundamentado na teoria dos *Checkpoints* Globais Consistentes (MOREIRA; SANTANA; SANTANA, 2005), utilizada em sistemas tolerantes a falhas, não fazendo uso de anti-mensagens para alcançar o sincronismo entre os processos.

Por sua vez, apesar da quantidade de protocolos, as ferramentas de auxílio para a

implementação de programas de simulação que utilizem estes protocolos são, de forma geral, específicas, oferecendo pouca flexibilidade para os usuários. Os usuários de simulação acabam usando aplicativos desenvolvidos para arquiteturas seqüenciais, que já estão bem consolidados no mercado, ou utilizando ferramentas para o desenvolvimento de programas paralelos e/ou distribuídos. Dentre estas ferramentas, destacam-se: ClassdescMP (STANDISH; MADINA, 2006), OOPS (*Object-Oriented Parallel System*) (SONODA; TRAVIESO, 2006), OODFw (*Object-Oriented Distributed Framework*) (DIACONESCU; CONRADI, 2002), SAMRAI (*Structured Adaptive Mesh Refinement Applications Infrastructure*) (WISSINK et al., 2001) e COOL (*Concurrent Object-Oriented Language*) (CHANDRA, 1995).

3. A modelagem do *framework*

Este artigo apresenta um *framework* para facilitar o desenvolvimento de programas de simulação que utilizem uma infra-estrutura distribuída através de uma máquina paralela ou, principalmente, um sistema distribuído. Um ambiente de simulação distribuída apresenta, basicamente, um programa de simulação, que foi implementado utilizando algum protocolo de sincronização, e ferramentas de comunicação, que permitem aos processos da aplicação executar em uma arquitetura paralela e/ou distribuída. Esta descrição permite estruturar os principais componentes em uma arquitetura de *software*, que é uma estruturação do *software* em camadas ou módulos. Neste contexto, um ambiente de simulação distribuída pode ser estruturado em uma arquitetura conforme ilustra a Figura 1.

Figura 1. Arquitetura em camadas de um ambiente de simulação distribuída

O primeiro nível, formado pela camada "Arquitetura Física", é responsável em manter as informações atualizadas da arquitetura física onde o programa de simulação será executado. O segundo nível, formado pela camada de comunicação, é responsável pelas trocas de informações realizadas durante a simulação. Estas trocas de informações são feitas através do envio e do recebimento de mensagens, entre os processos envolvidos na simulação, pelos canais de comunicação do sistema. Esta camada também deve garantir que toda mensagem enviada por um processo será recebida pelo processo receptor e que a ordem cronológica de envio será respeitada no recebimento.

O terceiro nível é formado pelo protocolo que é responsável em realizar a interface entre as camadas de comunicação e aplicação. Neste nível se encontram os protocolos apresentados neste texto. Esta camada é responsável em tratar os erros de causa e efeito a fim de garantir a consistência dos resultados obtidos pela simulação.

O quarto e último nível realiza a interface com o usuário permitindo ao mesmo tempo entrar com os modelos a serem simulados e realizar a coleta dos dados após a simulação.

Com este modelo em camadas, o *framework* alcança um alto nível de flexibilidade, permitindo combinar diversas estruturas de forma simples e eficiente. É possível, por exemplo,

utilizar uma implementação do protocolo *Time Warp*, utilizando a biblioteca de troca de mensagens PVM (*Parallel Virtual Machine*), ou uma implementação do protocolo *Rollback* Solidário, utilizando a biblioteca MPI (*Message Passing Interface*).

A seguir será apresentada a modelagem do *framework* em diagramas de classes e de seqüência da UML (*Unified Modeling Language*).

3.1 Diagrama de classes

Através da descrição das classes do modelo desenvolvido, os principais elementos fixos e flexíveis do *framework*, além de suas relações, serão apresentados e discutidos, com o propósito de facilitar a compreensão dos recursos disponíveis para o desenvolvimento de aplicações específicas.

O principal objetivo é proporcionar soluções para usuários que desejam realizar simulações de modelos discretos em ambientes distribuídos. São oferecidos recursos para a utilização de protocolos que garantem a integridade dos resultados da simulação, assim como a escolha de tais protocolos, em particular os otimistas: *Time Warp* e *Rollback* Solidário. Outro recurso oferecido é a possibilidade de se implementar a migração de processos e a troca dinâmica entre os protocolos.

A classe principal deste *framework* é a classe `Ambiente`. Esta é uma classe contêiner que define o ambiente principal que hospedará os principais objetos da aplicação do usuário, conforme pode ser visto na Figura 2. A classe `Ambiente` tem a função principal de abstrair a hierarquia apresentada na Figura 1, ou seja, separar a arquitetura física onde ocorrerá a simulação, o ambiente de passagem de mensagens que será utilizado, o protocolo de simulação distribuída e o modelo que será simulado.

A classe `Ambiente` é composta de quatro objetos principais: um objeto da classe `Protocolo`, um objeto da classe `Comunicacao`, um objeto da classe `Arquitetura` e outro objeto da classe `EstadoGeral`. As classes `Protocolo`, `Comunicacao` e `EstadoGeral` são classes abstratas para proporcionar maior flexibilidade para o usuário do *framework*. A classe `EstadoGeral` tem por finalidade armazenar informações gerais do comportamento da simulação, ou seja, manter as informações que se deseja analisar.

Além destes atributos, a classe `Ambiente` possui dois métodos principais: `PrepararAmbiente()` e `Executar()`. O primeiro método invoca os métodos de inicialização dos objetos das classes `Arquitetura` e `Protocolo` que, por sua vez, também envia a mensagem de preparação para o objeto da classe `Modelo`, escalonando os respectivos processos lógicos aos processadores, conforme estrutura definida na classe `Arquitetura`.

Toda a descrição lógica da arquitetura física do sistema fica sobe a responsabilidade da classe `Arquitetura`. São informações como o número de estações da rede e as características de cada um destes processadores. Estas informações são armazenadas em uma lista onde cada nó contém dados sobre: os recursos de processamento de cada estação, a quantidade de memória e outras informações necessárias para as tomadas de decisão a respeito do mapeamento dos processos envolvidos.

Destaca-se que as informações da arquitetura são passadas para o objeto através dos dados armazenados em um arquivo texto previamente configurado.

A classe `Modelo` é abstrata para permitir que modelos elaborados em ferramentas de modelagem diferentes como, por exemplo: Redes de Filas, *State Charts* e Redes de Petri, possam ser utilizados com as classes do *framework*. Para isto, será necessário sobrescrever o método abstrato `ConverteModelo()`, que tem a função de atualizar os atributos da classe, nas classes concretas descendentes.

Informações a respeito do modelo simulado serão tratadas na classe `Modelo` como um grafo dirigido. Há uma lista contendo informações sobre cada processo lógico como, por exemplo, a probabilidade de ser criado um novo evento a cada iteração da simulação, as probabilidades de que sejam criados eventos para outros processos lógicos, durante o tratamento

de um evento, e a probabilidade de que um evento termine sem gerar novas atividades.

Semelhante à classe `Arquitetura`, a classe `Modelo` recebe as informações pertinentes ao modelo que o usuário pretende simular através de um arquivo de configuração. Isso permite maior flexibilidade e ainda possibilita a simulação de diversos modelos sem a necessidade de escrever um programa para cada um deles.

Como existem diversos métodos para geração de números aleatórios, o objetivo da classe `NumeroAleatorio` é permitir que sejam implementados os algoritmos das distribuições de probabilidade necessárias à simulação dos modelos dos usuários. É também, desta forma, uma classe abstrata devido ao método `GerarProxNumero()`. Basicamente, a classe mantém um valor inicial (semente) e, a partir deste valor, este método deverá gerar a nova semente.

A classe `Comunicacao` representa as funções de comunicação existentes nos ambientes de troca de mensagens do sistema. De tal modo, a partir desta classe, podem-se criar classes concretas que implementem as primitivas de comunicação existentes nos ambientes de interesse do usuário. Esta classe não possui atributos, mas suas classes descendentes possuirão, basicamente, dois atributos: um vetor com os identificadores dos processos, que farão parte do ambiente de comunicação, e um objeto com a estrutura das mensagens que trafegarão na rede. Por sua vez, a classe abstrata `Mensagem` permite ao usuário do *framework* adaptar a estrutura desta classe para os dados específicos que ele deseja manipular durante a comunicação dos processos do seu programa de simulação.

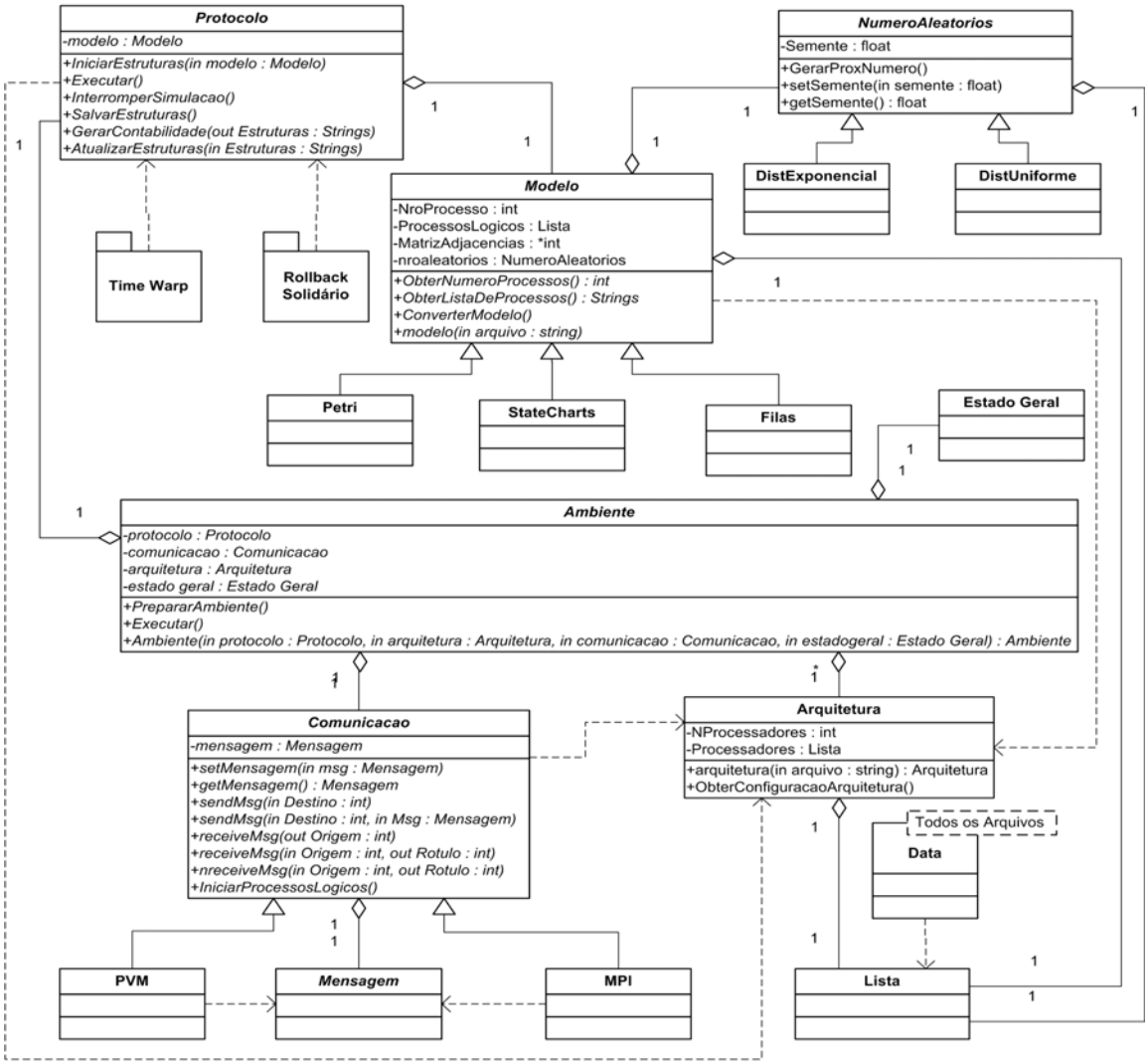

Figura 2. Diagrama de classes do *framework* proposto

A classe `Protocolo` possui mecanismos que simplificam a utilização dos protocolos de sincronização. Esta classe está associada aos componentes que compõem as estruturas de cada implementação de um protocolo. É também uma classe abstrata, pois para cada tipo de protocolo, será necessário criar uma classe descendente concreta. Esta classe descendente é o ponto de ligação do *framework* com o pacote que corresponde às classes que, efetivamente, descrevem o funcionamento do protocolo em questão. Isto pode ser visto através das classes `TW` e `RS` e respectivas associações com os pacotes "`Time Warp`" e "`Rollback Solidario`". O método `IniciarEstruturas()` prepara as estruturas do protocolo para a simulação. Este método, associado ao protocolo *Time Warp*, por exemplo, inicia as listas de eventos futuros, listas de mensagens e de anti-mensagens, além de realizar o salvamento do estado inicial de cada processo lógico. O método `InterromperSimulacao()` é um método que pode ser invocado externamente caso o usuário deseje verificar como está o andamento da simulação, ou seja, avaliar os dados parciais produzidos até um determinado instante. Além disso, este método pode ser utilizado para implementar um mecanismo de troca dinâmica de protocolos. Neste caso, porém, será necessário invocar o método `SalvarEstruturas()` para armazenar as respectivas informações. A partir deste ponto, a simulação é reiniciada com outro protocolo.

No diagrama principal do *framework* (Figura 2), há dois pacotes relacionados com a classe `Protocolo`: "`Time Warp`" e "`Rollback Solidario`". Estes pacotes são utilizados para permitir o controle de acesso a seus conteúdos, de modo que seja possível controlar as costuras existentes na arquitetura do sistema. Desta forma, a classe `Protocolo` realiza o gerenciamento dos pacotes correspondentes às implementações dos protocolos para simulação distribuída. Essa classe também funciona como interface entre estes pacotes e as demais classes do *framework*. Qualquer protocolo que possa ser implementado mantendo as regras de comunicação com a classe `Protocolo` pode se tornar um protocolo disponível no *framework* para utilização dos usuários em geral, até mesmo protocolos conservativos, como é o caso do protocolo CMB. O método abstrato `SalvarEstruturas()` tem a função de armazenar o estado das variáveis internas de gerenciamento do protocolo. Desta forma, quando a simulação necessitar ser retomada, estes dados serão utilizados para reiniciar a simulação do ponto onde ocorreu a interrupção. O método `GerarContabilidade()` retorna uma lista parametrizada de *strings* contendo informações a respeito do protocolo. No protocolo *Time Warp*, exemplos destas informações são: a quantidade de *rollbacks* realizados, o tamanho médio da fila de eventos futuros e o tamanho médio da fila de anti-mensagens, além dos estados atuais destas estruturas; os números de eventos realmente realizados (que não serão mais afetados por um *rollback*) e a quantidade de eventos desfeitos devido aos procedimentos de *rollback*. No *Rollback* Solidário, além destas informações, a lista acrescenta o número de *checkpoints* básicos e o núumero de *checkpoints* forçados realizados pelo método semi-síncrono adotado. Estas informações são essenciais para a implementação de um procedimento de troca dinâmica de protocolos. Além disso, estas informações possibilitam comparar o desempenho do protocolo corrente em tempo de execução.

O método `AtualizarEstruturas()` é um método que permite iniciar as estruturas do protocolo com dados previamente configurados que são passados através de uma lista de *strings*. Esta lista tem a mesma estrutura daquela retornada pelo método `GerarContabilidade()`. Através deste método é possível iniciar um protocolo com dados de uma simulação que foi parcialmente realizada anteriormente. Assim, este método será utilizado em um mecanismo de troca de protocolos, ou mesmo, para continuar uma simulação que foi interrompida anteriormente, permitindo a criação de mecanismos de persistência da simulação. Estes métodos devem ser tratados nas classes concretas descendentes da classe `Protocolo`.

3.2 O pacote *Time Warp*

Para que fique claro como devem ser tratados as implementações de cada protocolo,

este artigo inclui a modelagem do protocolo otimista *Time Warp*. A Figura 3 apresenta o diagrama de classes que está contido no pacote "`Time Warp`".

No diagrama da Figura 3, a classe `Processo` é a base do modelo da implementação do protocolo, pois representa o processo lógico da simulação, cujo código principal deve ser descrito no método `executar()`. A classe `Observador`, por sua vez, possui os métodos abstratos `EscalonarProcessos()` e `IniciarTrocaDeProtocolo()`, que deverão ser utilizados em mecanismos de troca dinâmica de protocolos, sendo que o método `IniciarTrocaDeProtocolo()` permite paralisar as estruturas no protocolo corrente e o método `FinalizarTrocaDeProtocolo()` inicia as estruturas no protocolo destino da troca. As classes `Processo` e `Observador`, possuem o método `ObterMensagem()` para facilitar a interação com o objeto da classe `Receptor`.

A classe `Estado` representa os atributos de cada processo da simulação. A separação destes atributos da classe `Processo` ocorre devido à necessidade de armazenar os estados durante o procedimento de *checkpointing*. Além disso, esta classe pode ser remodelada de acordo com a necessidade do usuário sem grandes impactos no sistema. A principal característica desta classe é a existência dos métodos `Serializar()` e `Desserializar()` que são responsáveis pela conversão dos atributos dos objetos da classe em uma cadeia de caracteres e vice-versa. Isto facilita o procedimento de *checkpointing* e a restauração dos respectivos estados durante um *rollback*.

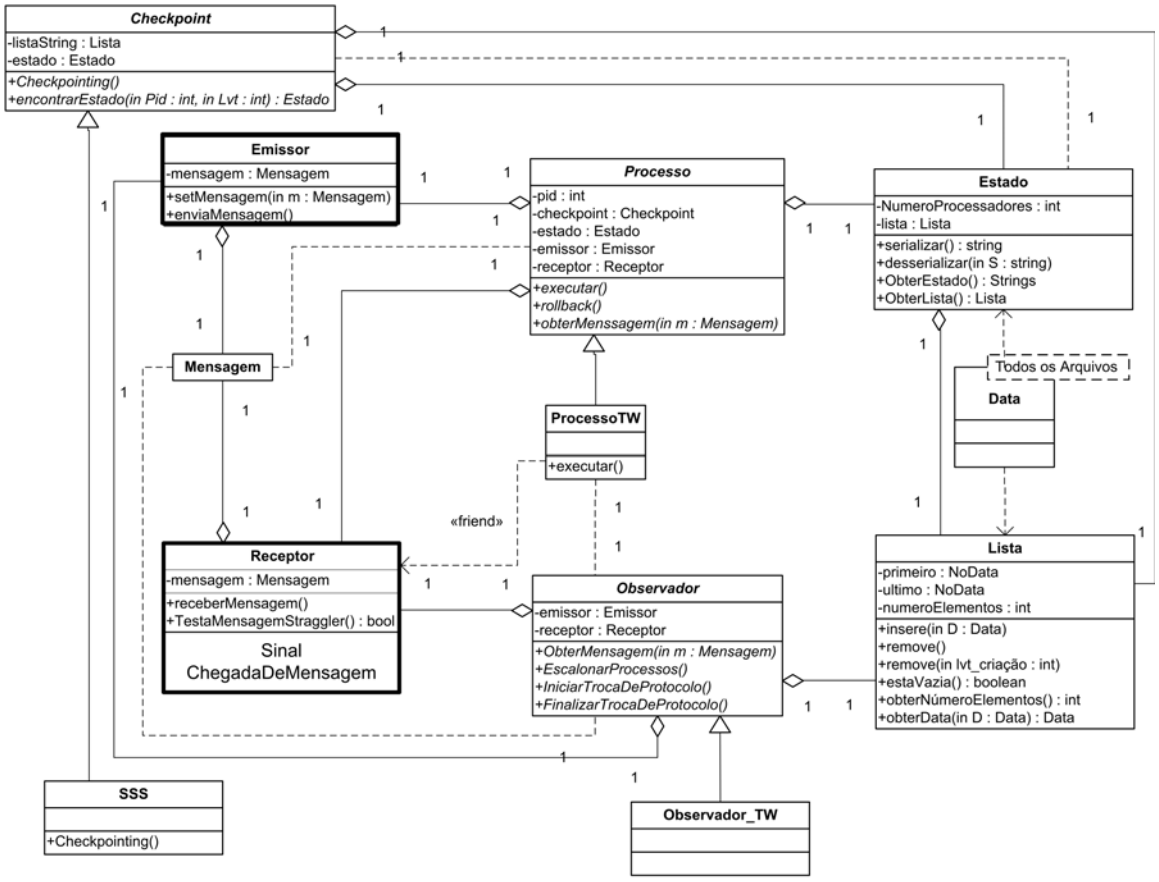

Figura 3. Diagrama de classes do protocolo *Time Warp*

As classes `Lista` e `Data` são classes que implementam as estruturas de dados necessárias para o gerenciamento das listas e filas de eventos futuros da simulação. Por conseguinte, a classe `Data` é uma classe-template, que é um elemento parametrizado. Os objetos

da classe `Data` devem ser instanciados de acordo com o tipo de dados que a classe `Lista` irá manipular durante a simulação.

O diagrama de classes apresenta duas classes ativas: `Emissor` e `Receptor`. Elas são responsáveis pela comunicação com as rotinas do nível 1 da arquitetura em camadas do ambiente de simulação. A existência destas classes especiais permite a implementação de linhas de controle (*threads*) independentes para o tratamento das mensagens do sistema, melhorando o desempenho do programa de simulação. Além disso, a classe `Receptor` é modelada como classe amiga da classe `ProcessoTW` para facilitar a comparação dos tempos lógicos das mensagens com o estado do objeto, ou seja, identificação de mensagens *stragglers* (mensagens que provocam erros de causa e efeito nos processos lógicos).

Na modelagem, a comunicação entre objetos ativos é descrita utilizando eventos, sinais e mensagens. Na classe `Receptor`, o sinal `ChegadaDeMensagem` alerta para a chegada de uma mensagem na rede de comunicação. As duas classes ativas se relacionam com a classe `Mensagem` que contém a estrutura da mensagem de comunicação.

Finalmente, há a classe abstrata `Checkpoint` cujas classes concretas descendentes são implementações dos mecanismos de gerenciamento de memória como, neste caso, o *Sparse State Saving* (SSS).

3.3 Uso do *Framework*

Esta seção discute a utilização do *framework* por parte do usuário para o desenvolvimento de aplicações de simulação.

Quando o método `PrepararAmbiente()`, da classe `Ambiente`, é invocado (Figura 4), a primeira ação realizada é o envio da mensagem `IniciarEstruturas()` ao objeto da classe `Protocolo`. Esta mensagem faz com que o objeto da classe `Protocolo` obtenha a quantidade de processos lógicos do modelo a ser simulado e, em seguida, a lista contendo as respectivas informações de cada processo. Com estas informações, o método `IniciarEstruturas()` é encerrado, retornando a configuração do modelo a ser simulado. O método `PrepararAmbiente()` envia a mensagem `ObterConfiguracaoArquitetura()` para o objeto `Arquitetura` que retorna com a estrutura disponível para a execução do programa de simulação. Finalmente, o método `IniciarProcessosLogicos()`, da classe `Comunicacao`, é invocado. Este método irá iniciar a máquina virtual.

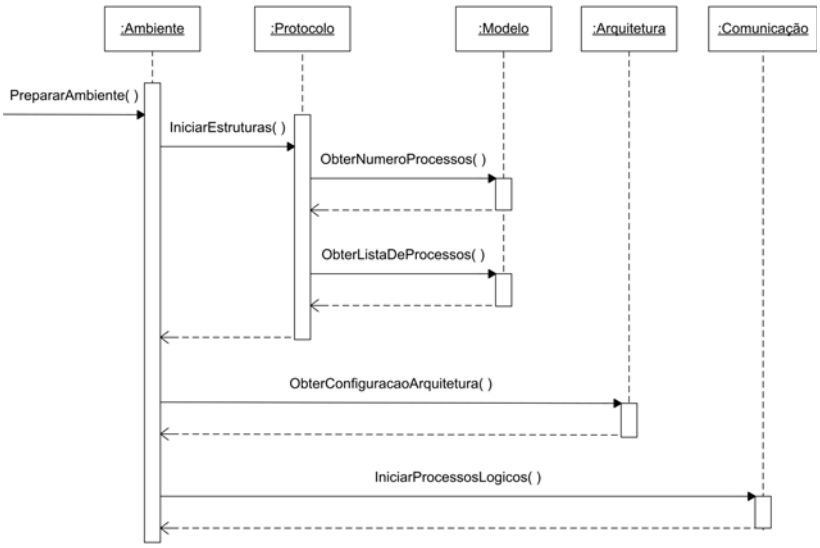

Figura 4. Diagrama de seqüência ilustrando a preparação do ambiente para a simulação

Uma vez que o ambiente está preparado, a simulação pode ser iniciada. Basicamente, o método `Executar()`, da classe `Ambiente`, invocará o método `Executar()` da classe `Protocolo`. Este último irá iniciar cada processo lógico da simulação e aguardar que cada um destes processos execute o número de iterações definidas no modelo. Quando a execução de cada processo lógico terminar, o método `Executar()` encerra sua chamada atualizando as variáveis que contabilizam os dados da simulação. Desta forma, o método `GerarContabilidade()` pode ser invocado para que os resultados da simulação sejam apresentados e utilizados pelo usuário da simulação.

A Figura 5 apresenta um exemplo de código que ilustra a facilidade para se desenvolver uma aplicação distribuída de simulação, usando as classes do *framework*. Trata-se de uma função principal em C/C++, linguagem utilizada na implementação do *framework*. Inicialmente, é instanciado um objeto da classe `Modelo`, denominado `ModeloParaSimulacao` (linhas 3 e 4). Em seguida, o protocolo *Time Warp* é escolhido para sincronizar os processos através da instanciação do objeto `TWProtocol` (linhas 5 e 6). A biblioteca de troca de mensagens escolhida é o PVM (linha 7) e a estrutura física da rede de computadores utilizada é descrita através da especificação contida no arquivo "`Arquitetura.dat`" (linhas 8 e 9). Este arquivo é utilizado pela classe `Arquitetura`. Após a declaração destes objetos, a classe `Ambiente` pode ser instanciada. Isto ocorre com a chamada do construtor que recebe como parâmetros os objetos `SistemaDistribuido`, `TrocaMensagem` e `TWProtocol` (linhas 10,11,12 e 13). Com os objetos preparados, a simulação pode ser iniciada. Para isso, é preciso invocar o método `PrepararAmbiente()` da classe `Ambiente` (linha 14). Este método ira invocar os métodos necessários para iniciar os processos lógicos da simulação e mapear estes processos nas estações apropriadas, de acordo com a descrição contida na arquitetura e com os critérios de escalonamento utilizados pelo protocolo. Após este passo, a simulação inicia com a chamada ao método `Executar()` da classe `Ambiente` (linha 15).

```
1  #include "framework.h"
2  int main() {
3    Modelo *ModeloParaSimulacao =
4      new Modelo ("Modelo.dat");
5    Protocolo *TWProtocol =
6      new TimeWarp(ModeloParaSimulacao);
7     Comunicacao *TrocaMensagem = new PVM();
8     Arquitetura *SistemaDistribuido =
9       new Arquitetura ("Arquitetura.dat");
10    Ambiente *AmbienteSimulacao =
11      new Ambiente(SistemaDistribuido,
12                   TrocaMensagem,
13                   TWProtocol);
14    AmbienteSimulacao->PrepararAmbiente();
15    AmbienteSimulacao->Executar();
16    TWProtocol->GerarContabilidade("Saida.dat");
17    return 0;
18 }
```

Figura 5. Exemplo de utilização do *framework* para a criação de um programa de simulação

4. Mecanismo de troca dinâmica de protocolo

Para que a troca dinâmica entre protocolos seja realizada com sucesso é necessário que exista uma classe que decida quando deve ocorrer a alteração dos protocolos. No *framework* apresentado, a classe responsável em realizar esta mudança é a classe `Protocolo`. Para também desempenhar esta função, a classe `Protocolo` deve equacionar o problema do desempenho da simulação, por isto, ela deve receber, das classes `Observador` dos protocolos implementados no *framework*, as informações referentes à ocorrência de *rollbacks* durante a simulação. Com estas informações, a classe `Protocolo` decide quando deve ser realizada a troca global de protocolos em todos os processos da simulação.

A troca deve ser realizada com segurança para garantir que *rollbacks* possam ser realizados corretamente após a mudança do protocolo. O problema para mudar o protocolo é a possibilidade de ocorrência de um *rollback* que force a recuperação do sistema para um ponto da simulação em que o protocolo que tratou tais eventos não seja o protocolo corrente. Como os protocolos, de forma geral, tratam erros de causa e efeito de forma diferente entre si, a ocorrência de uma situação assim poderia, por exemplo, obrigar o protocolo *Time Warp* a realizar um procedimento de *rollback* sem a utilização de anti-mensagens. Desta forma, para que a troca seja possível, será necessário garantir que todo passado histórico do protocolo, antes da troca, não será utilizado. E a maneira de atingir parte deste objetivo é utilizando o cálculo do GVT (*Global Virtual Time*). Porém, ainda assim há, durante a troca do protocolo, probabilidade de surgirem mensagens *stragglers*. Para esta situação, a solução é permitir que os protocolos coexistam por um determinado período, com o propósito de fornecer dados suficientes ao novo protocolo responsável pela simulação.

Definir o período de coexistência dos protocolos que estarão envolvidos na troca é uma função da classe `Protocolo`. Para isto, deve-se estipular o tempo mínimo necessário para a transição. Assim, quando a classe `Protocolo` decidir trocar o protocolo corrente, ela deve proceder o cálculo do GVT. Durante este processo, a classe `Protocolo` deve obter o LVT máximo do sistema definindo a fronteira final da simulação híbrida, ou seja, quando o LVT máximo for o GVT da simulação o novo protocolo poderá assumir a simulação, pois o sistema já possuirá todos os dados necessários para realizar uma recuperação do sistema, se necessário.

Todo gerenciamento da troca dinâmica é realizado pelo processo `Observador`, através da classe `Observador`. Qualquer processo lógico, através da classe `Protocolo`, pode identificar a viabilidade da troca de protocolos. Quando isto ocorre, o processo lógico envia um sinal para o processo `Observador` solicitando a mudança de protocolo. Neste momento, o processo `Observador` analisa se o procedimento pode ocorrer ou não e gerencia o procedimento de troca. Por exemplo, Moreira (MOREIRA, 2005) apresentou um estudo comparativo entre os protocolos *Time Warp* e *Rollback* Solidário. Neste estudo, foi demonstrado que durante o tratamento de um *rollback* o protocolo *Time Warp* apresenta um desempenho superior em relação ao protocolo *Rollback* Solidário quando o número de processos envolvidos no *rollback* não ultrapassa 3. Assim, uma política de troca de protocolos pode ser desenvolvida a partir do número de processos envolvidos em cada *rollback*.

5. Conclusões

O uso deste *framework* favorece o aumento do desempenho das aplicações de simulação, pois pesquisadores que não possuem conhecimento na área de sistemas distribuídos podem realizar simulações distribuídas obtendo resultados com maior eficiência. Além disso, o tempo necessário para a criação deste tipo de aplicação diminui significativamente.

O *framework* implementa os níveis 1 e 2 da arquitetura em camadas da Figura 1. Entretanto, a criação de uma interface amigável com o usuário (nível 3) ou a união com outras ferramentas de modelagem pode facilmente ser realizada, produzindo uma ferramenta automatizada de modelagem e geração de programas paralelos e/ou distribuídos para simulação.

Finalmente, destaca-se que este *framework* é uma ferramenta de apoio aos profissionais que necessitam desenvolver programas eficientes de simulação. Desta forma, a partir dele, vários trabalhos podem ser desenvolvidos, tanto na área de *frameworks* para aplicações distribuídas, quanto na área de protocolos de sincronização. Além disso, a modelagem desenvolvida neste trabalho oferece vários recursos para implementação de outros *frameworks*.

Agradecimentos

Este trabalho foi parcialmente financiado por CNPq, CAPES, FAPEMIG e FINEP.

Referências

Boukerche, A. and Tropper, C. (2001). Local versus global lookahead in conservative parallel simulations. *Parallel Computing*, 27(8):1033–1055.

Bryant, R. E. (1977). Simulation of packet communication architecture computer systems. Technical report,Massachussets Institute of Technology,Massachussets. MIT-LCS-TR-188.

Cai, W. and Turner, S. J. (1990). An algorithm for distributed discrete-event simulation – the "carrier null message" approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):3–8.

Chandra, R. (1995). *The COOL Parallel Programming Language: Design, Implementation and Performance*. PhD thesis, Computer Science Department, Stanford University.

Chandy, K. M. and Misra, J. (1979). Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452.

Diaconescu, R. and Conradi, R. (2002). A Data Parallel Programming Model Based on Distributed Objects. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 455–460.

Ferscha, A. (1995). Probabilistic adaptive direct optimism control in time warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 120–129.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*. JohnWiley & Sons, New York.

Fujimoto, R. M. (2003). Distributed simulation systems. In *Proceedings of the 2003 Winter Simulation Conference*, pages 124–134.

Iskra, K. A., Albada, G. D. V., and Sloot, P. M. A. (2003). Time warp cancellation optimisations on high latency networks. In *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 128–135.

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425.

Ji, A., Zhou, J., Takai, M., Martin, J., and Bagrodia, R. (2004). Optimizing parallel execution of detailed wireless network simulation. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 162–169.

Liu, L. Z. and Tropper, C. (1990). Local deadlock detection in distributed simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 64–69.

Misra, J. (1986). Distributed discrete-event simulation. *ACM Computing Surveys*, pages 39–65.

Moreira, E. M. (2005). *Rollback Solidário: Um novo protocolo otimista para Simulação Distribuída*. Tese Doutorado, Universidade de São Paulo.

Moreira, E. M., Santana, R. H. C., and Santana, M. J. (2005). Using consistent global checkpoints to synchronize processes in distributed simulation. In *Proceedings of the 9th The 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pages 43–50.

Pegden, C. D., Shannon, R. E., and Sadowski, R. P. (1995). *Introduction to Simulation using SIMAN*. McGraw-Hill International Editions, New York, 2nd edition.

Sonoda, E. and Travieso, G. (2006). The OOPS Framework: High Level Abstractions for the Development of Parallel Scientific Applications. In *OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 659–660, New York, NY, USA. ACM Press.

Srinivasan, S. and Reynolds, P. F. (1998). Elastic time. *ACM Transactions on Modeling and Computer Simulation*, 8(2):103–139.

Standish, R. and Madina, D. (2006). Classdesc and Graphcode: support for scientific programming in C++. *submitted to SIAM Journal of Scientific Computing*.

Taylor, S. J. E., Bruzzone, A., Fujimoto, R., Gan, B. P., Strassburger, S., and Paul, R. J. (2002). Distributed simulation and industry: Potentials and pitfalls. In *Proceedings of the 2002 Winter Simulation Conference*, pages 688–694.

Wissink, A. M., Hornung, R. D., Kohn, S. R., Smith, S. S., and Elliott, N. (2001). Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing SC'01*.

Zeng, Y., Cai, W., and Turner, S. J. (2004). Batch based cancellation: A rollback optimal cancellation scheme in time warp simulations. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 78–86.